Date __/__/____

Saathi

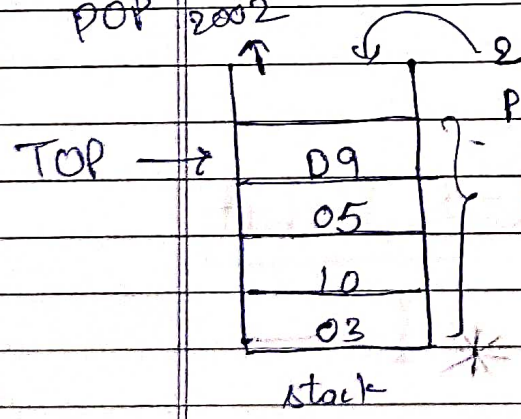
Stack, Queue
Dequeue, Priority Queue

Stack → It is a type of data structure used to solve many problems in a less amount of time because it follows "LIFO" method "Last In First Out" and there is only ~~side~~ one side for entry & to exit from.

Data kept on the top of another.
The element which is pushed in a stack will access lastly.

Operation → (1) PUSH → To add an element in the stack. the new element will become the top most element of the stack.

(2) POP → To remove an element from the stack. (Topmost)
(3) TOP → To get the value of the ~~po~~ topmost element.
(4) Size → To return the size of stack.
(5) EMPTY → Boolean function return True or False whether the stack is empty or not.



Saathi

Date __/__/____

Monotonic Stack

The term Monotonic means either increasing or decreasing.

Monotonic Stack are of two types :-

(1) Increasing
(2) Decreasing

(1) **Increasing Monotonic Stack** :- The elements keep on increasing as we move towards the bottom of the stack. The top most element is the lowest in value. The highest value element is the bottom most element or first inserted element.

(2) **Decreasing Monotonic Stack** :- Vice Versa of "IMS"

13
27
43
84

Increasing Montonic

84
43
27
13

Decreasing Monotonic.

Saathi

Date ___/___/_____

Stack Implementation → Method → (i) Array
(ii) Linked List.

Array Implementation

(i)

```
class Stack {
    public:
    int *arr;
    int top;
    int size;

    Stack (int size) {
        this -> size = size;
        arr = new int[size];
        top = -1; }

void push(int e) {
    if(size - top > 1) {
        top++;
        arr[top] = e;
    } else {
        cout << "Stack Overflow" << endl; }
}

void pop() {
    if(top >= 0) {
        top--;
    } else {
        cout "Stack overflow" << endl;
    } }
```

Page No. ____

Saathi

Date ___ / ___ / _____

```
int topfeek(){
    if (top>=0){
        return arr[top];}
    else{
        cout<<"Stack is empty"<<endl; return -1;
    }
    return -1;}

bool isEmpty(){
    if (top==-1){
        return true;
    }
    else{
        return false;}}
};
```

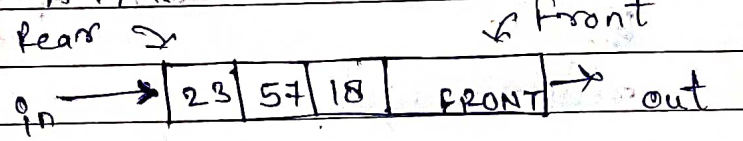
{ Linked List Implementation:- }.

Date __/__/____

* Queue

Saathi

It is a type of Data Structure which follows FIFO (First in First Out) technique for example →
The line of people who wait for their turn to go inside ATM.



Important Rule :-

(1) The data kept one after another. It is shown in the above that 18 is the front element & 23 is in Rear side.

(2) It is based on FIFO.

(3) The different operations in the queue are :-

- PUSH → The size of the queue increases by 1 when element is pushed inside in it. The front element will not be changed.
- POP → Pop means to remove an element from the queue. Obviously the front most element will only be removed. The size of the stack decreases by 1.
- FRONT → The element which is entered first is shown at Front, to get the value of the front most element.
- Size → To get the size of queue
- EMPTY → Boolean value. If empty returns true else false

Time complexity of all the above operations is constant O(1)

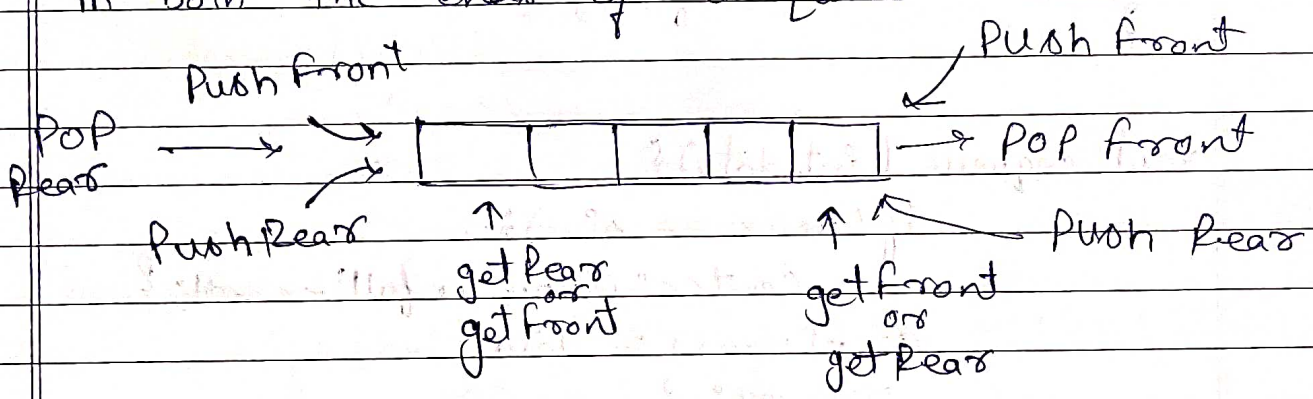
Date __/__/____

Saathi

"Dequeue"

Doubly ended Queue

It is same as of Queue but the difference is that it supports push & pop & getfront & getrear in both the ends of a queue



Operations are almost same, new operations.

Push Rear → To push element in Rear side of queue
Pop Rear → To pop " " " " " "
getRear → To get the Rear element or last entered element.

Implementation :- Using Array;

```
class queue{
    int front, rear, *arr, size
public:
    Queue(){
        size = 10001;
        arr = new int[size];
        front = 0 = rear; }
```

Types of Queue

(1) Dequeue
(2) Circular queue
(3) Priority queue - Ascending / Descending
(4) Input restricted
(5) Output restricted

Saathi

Date __/__/____

```
bool isEmpty(){
    if (rear == front){
        return true; }
    else return false;
}

void enqueue (int data){
    if (rear == size){
        cout<<"Queue is full"<<endl; }
    arr[rear] = data;
    rear++; }

int dequeue(){
    if (front == rear){
        return -1; }
    else {
        int ans = arr[front];
        arr[front] = -1;
        front++;
        if (front == rear){
            front = 0;
            rear = 0; }
        return ans; } }

int front(){
    if (front == rear){
        return -1; }
    return arr[front]; } };
```

Page No.

Date __/__/____

Saathi

Implementation of Dequeue:

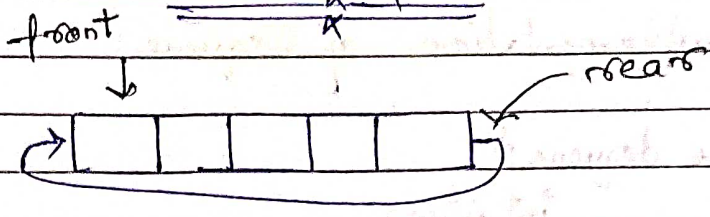
```
class dequeue {
    int size;
    int *arr, front, rear;
  dequeue(int n){
      size=n;
      arr = new int[n];
      front = rear = -1 ; }
bool pushF (int x){
    if ( front==0 && rear==size-1)||(rear==(front-1)%(size-1))){
        return false; }
    else if (front==-1) { front = rear = 0; }
    else if (front==0) { front=n-1; } // rear!=size-1
    else { front--; }
      arr [front] = x;
      return true;
bool pushR ( int x) {
    if ( front==0 && rear==size-1)||(rear==(front-1)%size-1)
      { return false; }
    else if ( front==-1) {
        front = rear=0; }
    else if (rear== size-1 && front!=0){
        rear=0; }
    else { rear++; } arr[rear] = x;
    return true; }
int popRear(){
        if(front==-1){ return -1; }
    int ans = arr[rear]
    arr[rear] = -1;
    if(front==rear){ front= rear = -1; }
    else if (rear==0) { rear= size-1; }
    else { rear--; } return ans;
```

Page No.

Date __/__/____

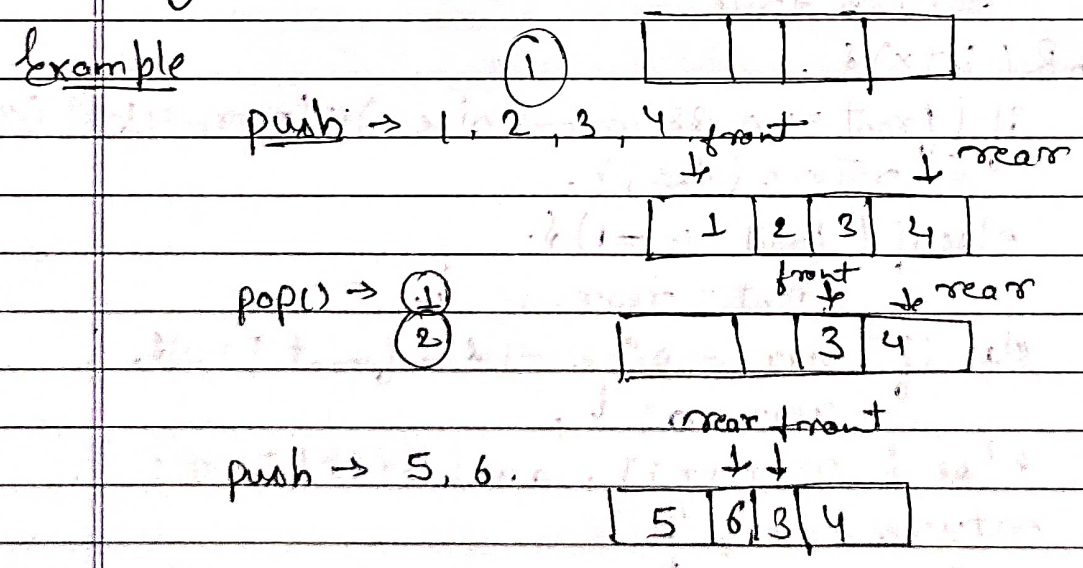
Saathi

Circular Queue



In circular queue the front & the rear pointer will not get out of bound in this type of queue for example. if rear is pointing towards the last position of the queue which is (size-1) then if the first entered element are poped out there is some space to store the new element then in that case the rear pointer will points to the 0th location and will push the element easily. Same for front it keeps pointing to the element untill it reaches to (size-1) then again revert back to 0th location. This is also called maintain cyclic nature in queue.

Example



this is called cyclic queue.

Date ___/___/_____

Saathi

Implementation

```
class Circularqueue {
    int *arr;
    int front, rear, size;
public:
    CircularQueue(int n) {
        size = n;
        arr = new int[size];
        front = rear = -1; }

bool enqueue(int value) {
    if((front == 0 && rear == size-1) || (rear == (front-1)%(size-)));
        {return false; }.
    else if(front == -1) {. front = rear = 0; }.
    else if(rear == size-1 && front != 0) { rear = 0; }.
    else { rear++ }.
    arr[rear] = value;
    return true; }.

int dequeue(){.
        if(front == -1) {.
            return -1; }.
        int ans = arr[front];
        arr[front] = -1;
        if(front == rear) {. front = rear = -1; }
        else if(front == size-1) front = 0;
        else front++;
        return ans;
    }
};
```

Priority Queue → It uses Heap concept we will study it in later chapters.

Saathi

Date ___/___/_____

Input Restricted Queue → Insert is allowed only from one side ie 'Rear' but pop /output is available on both side.

operation → push_back()
pop_front(), pop_back()

Output Restricted Queue → Insert is allowed on both side but output is restricted which is front

operation
push_front()
push_back()
pop_front()

Linked List Implementation of Queue.